BMED6517 Midterm Project Report

Mengyang Liu

October 2022

1 Methodology

1.1 Workflow Overview

Our method mainly focus on fin-tuning existing large-scale pre-trained deep learning models and apply it to the challenge, since it has been empirically acknowledged that such pre-trained model can capture some common features like edges and shape, it could save us a lot of time by fine-tuning. Unlike the baseline model provided in the notebook, the input of our method use the whole dataset as the input, which contains 1392 slices in total. The training dataset contains 1076 slices of 256 * 256 single channel gray scale image, the corresponding label is a image with same size, which we called mask. We have 116 slices for validation dataset and 200 slices for test dataset. Figure. 1 is a visualization of some samples in the training set.

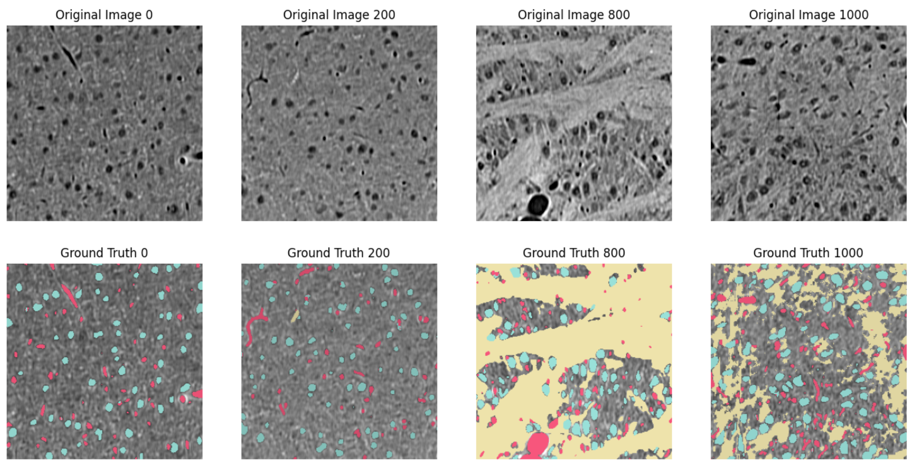

Figure 1: Visualization of some samples from training set

Our method will create an deep convolutional neural network with encoder-decoder architecture. First our model encode the image of size $H \times W$ into some latent space

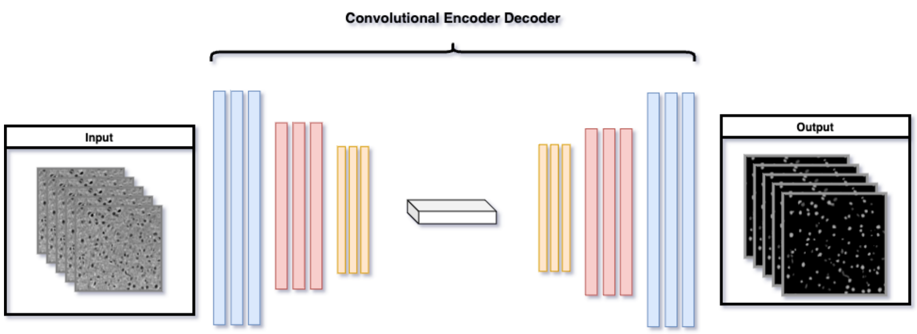

Figure 2: Workflow of the proposed framework

R^d, and then the decoder will use the representation to do up-sampling and output a matrix P with size of $C \times H \times W$. Here C is that class number, H and W are the height and width of image respectively, where P[c, i, j] is the probability that the pixel at position i, j belong to class c. Then we use functions like `torch.argmax` to get the predicted class. Our general workflow can be formulate as the diagram shown in Figure. 2.

2 Model Fitting and Hyperparameter Fine-tuning

2.1 Data Preprocessing and Statistics

In our method, we applied some augmentation before loading images into our model. Specifically, we adopt some augmentation method provided by torchvision like RandomVerticalFlip, RandomHorizontalFlip, RandomRotation. **The augmentation only applies to training dataset.** Figure. 3 shows some augmentation results.

	Train	Val	Test
BG pixels	39665515	4101022	6591602
BV pixels	2724278	243515	456854
cell pixels	4448051	532764	820941
axon pixels	23678892	2724875	5237803
Total pixels	1076 * 256 * 256	116 * 256 * 256	200 * 256 * 256
std	31.018	31.699	33.324
mean	123.24	127.8	130.48

Table 1: Dataset Statistics

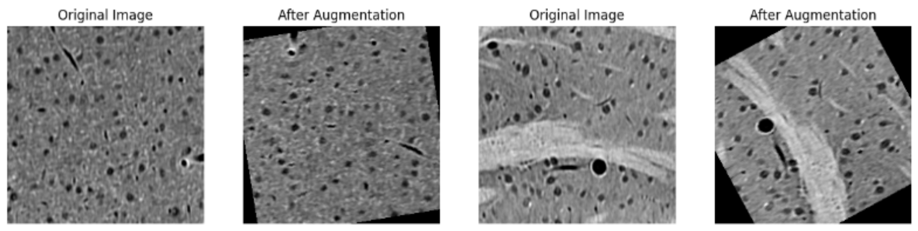

Figure 3: Augmentation comparsion

2.2 Model and Loss Function

After data preprocessing, we will load images into our models. Our method mainly adopted two kinds of model. **UNet**[1] and **LinkNet** [2]. UNet is a model family which is popular for Medical Imaging Segmentation task. LinkedNet tries to connect encoder and decoder directly and try to have a fusion on the corresponding feature map, which have faster speed and good performance. For loss function, we mainly use two functions, **Cross Entropy Loss** and **Focal Loss**. Cross Entropy Loss is a popular loss function for multi-class classification task, but naively apply it to a highly imbalanced class may not be a good idea. We may need to configure the weight for each class manually. Focal loss is built based on cross entropy loss, it works better than cross entropy when working with highly imbalanced data. Eq. 1 and Eq. 3 are formula for Cross Entropy Loss and Focal Loss respectively.

$$CE(p_t) = -\alpha_t \log(p_t) \tag{1}$$

$$\alpha_t = \begin{cases} \alpha & y = 1 \\ 1 - \alpha & \text{otherwise} \end{cases} \tag{2}$$

$$FL(p_t) = -(1 - p_t)^\gamma \log(p_t) \tag{3}$$

2.3 Evaluation and Analysis

For all of our model, we adopted ResNet with ImageNet weight as the encoder. **Note, we will call model as Model+Backbone + Loss type, for example. UNet18 + Focal means we use UNet with ResNet18 as the encoder, and we use Focal loss as the loss function.** First we use 5e-2 as the learning rate to train UNet18 + CE loss and LinkedNet18 + CE Loss. However, we found that the model cannot converge well. We just list the macro accuracy in **Table**. 2 for reference. We also think imbalanced data distribution could also be a reason, thus we assign more weight to class with fewer sample, the weightes are [0.1, 0.3, 0.3, 0.2].

Model setting	Precision	Recall	F1-Score
UNet18 + CE (lr=5e-2)	0.61	0.63	0.62
LinkNet18 + CE (lr=5e-2)	0.77	0.75	0.74
UNet18 + CE (lr=3e-2)	0.77	0.67	0.65
LinkNet18 + CE (lr=3e-2)	**0.81**	**0.83**	**0.83**

Table 2: Comparison of Macro Performance Score with Different Learning Rate

Then we kept the same learning rate and try model with Focal Loss. The result is shown in **Table**. 3. We found model with Focal loss get performance improved a lot, thus, we decide to move from cross entropy loss to Focal loss, and we want to try different encoders (resnet18, resnet34, resnet50) and explore that whether large model can benefit our segmentation task.

Model setting	Precision	Recall	F1-Score
UNet18 + Focal (lr=3e-2)	**0.89**	**0.88**	**0.89**
LinkNet18 + Focal (lr=3e-2)	**0.89**	**0.88**	0.88
UNet18 + CE (lr=3e-2)	0.77	0.67	0.65
LinkNet18 + CE (lr=3e-2)	0.81	0.83	0.83

Table 3: Comparison of Macro Performance Score between CE Loss and Focal Loss

Model setting	Precision	Recall	F1-Score	GPU Memory Usage
UNet18 + Focal	0.89	0.88	**0.89**	10544
UNet34 + Focal	0.87	**0.89**	0.88	11412
UNet50 + Focal	0.86	0.81	0.83	16204
LinkNet18 + Focal	0.89	0.88	0.88	7956
LinkNet34 + Focal	**0.91**	0.85	0.87	8820
LinkNet50 + Focal	0.90	0.85	0.86	15622

Table 4: Macro Performance Score with GPU Memory Usage

Model setting	Precision	Recall	F1-Score
UNet18 + Focal	**0.94**	**0.94**	**0.94**
UNet34 + Focal	**0.94**	0.93	0.93
UNet50 + Focal	0.88	0.88	0.87
LinkNet18 + Focal	**0.94**	0.93	0.93
LinkNet34 + Focal	**0.94**	**0.94**	**0.94**
LinkNet50 + Focal	0.92	0.92	0.92

Table 5: Weighed Performance Score

Model setting	Precision	Recall	F1-Score
UNet18 + Focal (Train, Macro)	0.91	0.91	0.91
UNet18 + Focal (Train, Weighted)	0.97	0.97	0.97
UNet18 + Focal (Validation, Macro)	0.88	0.88	0.88
UNet18 + Focal (Validation, Weighted)	0.92	0.92	0.92
UNet18 + Focal (Test, Macro)	0.89	0.88	0.88
UNet18 + Focal (Test, Weighted)	0.94	0.93	0.93

Table 6: LinkNet18 Performance Score

	Precision	Recall	F1-Score
BG	0.98	0.98	0.98
BV	0.83	0.84	0.84
cell	0.92	0.87	0.90
axon	0.91	0.95	0.93

Table 7: LinkNet18 Training Performance

	Precision	Recall	F1-Score
BG	0.97	0.90	0.93
BV	0.75	0.82	0.78
cell	0.93	0.83	0.88
axon	0.87	0.97	0.92

Table 8: LinkNet18 Validation Performance

	Precision	Recall	F1-Score
BG	0.97	0.93	0.95
BV	0.74	0.82	0.78
cell	0.92	0.81	0.87
axon	0.92	0.97	0.94

Table 9: LinkNet18 Test Performance

From **Table**. 4 and **Table**. 5, we can see that with slightly large model (resnet34), LinkedNet have some trivial improvement and UNet didn't benefit from it. Generally, resnet18 and resnet34 both can work well. But for resnet50, there's a obvious performance drop, which may imply that the model is too complicated for the dataset. Over-fitting may happen.

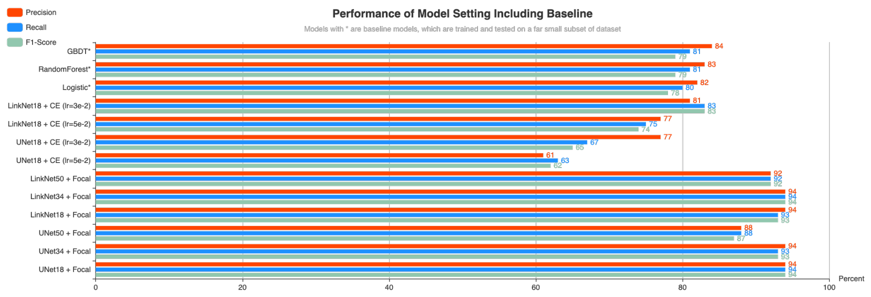

Figure 4: Performance Visualization

In Figure. 4, we compared the performance of all setting we tried and we also listed the performance of three baselines (Logistic Regression, Random Forest and Gradient Boost Descent Tree), though these baselines are trained on a far small subset of the dataset we used in our experiments. Our method, **lr=3e-2, Focal loss with UNet / LinkNet** have far better performance than the baselines.

3 Conclusion

The improvement is mainly from the powerful deep learning feature extractor. It enable model to extract rich spatial feature from images. The baseline only have a few feature provided by the identity filter which lose a lot information compared to the feature extracted by deep learning model. During tuning the hyperparameters, I tried to decrease the learning rate when the model doesn't converge well and I notice may need to try a lower learning rate and considering the imbalanced dataset, it's better the use weight to regularize the cross entropy loss. It did work and the performance improved nearly 10 percents. After that, I consider may be using a loss dedicated for imbalanced dataset, cause manual weight is too subjective. Thus, I tried Focal loss and it worked very well and bring my weighted accuracy to 0.94, which is a huge improvement. But when I tried to use larger model to extract more informative features, it didn't work. The lesson I learned is that we should keep in mind the nature of the dataset, the imbalanced dataset should be carefully treated, we should use specific loss or training method to deal with it. If I have more time, I may try to extract the features from some intermediate kernels and use some shallow learning method to do the classification.

	Precision	Recall	F1-Score
BG	0.93	0.97	0.95
BV	0.75	0.82	0.78
cell	0.92	0.81	0.86
axon	0.97	0.92	0.94

Table 10: Unet18 Test Performance

	Precision	Recall	F1-Score
BG	0.95	0.95	0.95
BV	0.91	0.57	0.70
cell	0.82	0.90	0.86
axon	0.94	0.96	0.95

Table 11: LinkNet34 Test Performance

	Precision	Recall	F1-Score
BG	0.97	0.92	0.94
cell	0.79	0.73	0.76
BV	0.78	0.93	0.85
axon	0.92	0.97	0.95

Table 12: Unet34 Test Performance

	Precision	Recall	F1-Score
BG	0.95	0.92	0.94
BV	0.91	0.61	0.73
cell	0.83	0.90	0.86
axon	0.91	0.96	0.93

Table 13: LinkNet50 Test Performance

	Precision	Recall	F1-Score
BG	0.89	0.89	0.89
BV	0.82	0.64	0.71
cell	0.88	0.81	0.84
axon	0.86	0.89	0.88

Table 14: Unet50 Test Performance

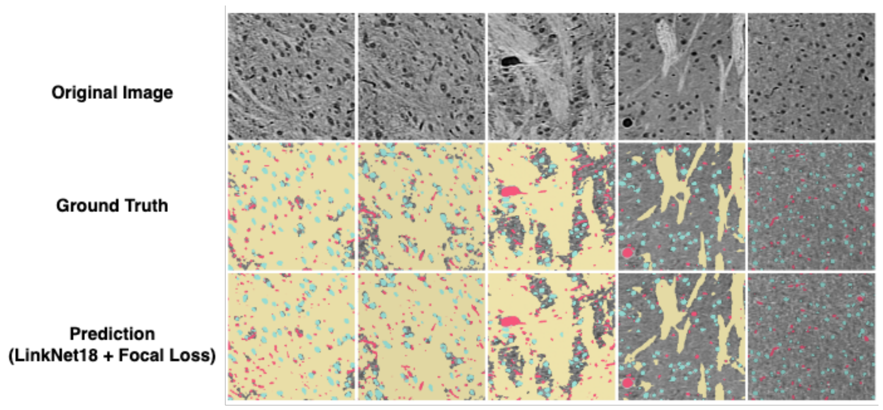

Figure 5: Comparison of Prediction and Ground Truth

References

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[2] Abhishek Chaurasia and Eugenio Culurciello. Linknet: Exploiting encoder representations for efficient semantic segmentation. In *2017 IEEE Visual Communications and Image Processing (VCIP)*, pages 1–4. IEEE, 2017.

```python
# config.py
import os
CLASS_NUM = 4
DIM = 2
LOSS_MODE = 'multiclass'
DEBUG = True

RESULT_DIR = 'results'
os.makedirs(RESULT_DIR, exist_ok=True)


MODEL = 'unet'
ENCODER = 'resnet34'
LOSS_TYPE = 'focal'
ENCODER_WEIGHT = 'imagenet'
GPU_ID = 4
# MODEL = 'fpn'
# encoder = 'resnet34'
# encoder_weights = 'imagenet'
# data.py
import numpy as np
import pytorch_lightning as pl
from torch.utils.data import DataLoader, Dataset
from torchvision.transforms import ToTensor, Compose, RandomHorizontalFlip,
RandomVerticalFlip, RandomRotation
from config import *
import matplotlib.pyplot as plt

def get_task_data_in_numpy():
    dim = DIM
    class_num = CLASS_NUM
    BASE_DIR = 'downloads/' + f'task2_{dim}D_{class_num}'
    images = {
        'train': np.load(BASE_DIR +'classtrainimages.npy').astype(np.uint8),
        'val': np.load(BASE_DIR +'classvalimages.npy').astype(np.uint8),
        'test': np.load(BASE_DIR +'classtestimages.npy').astype(np.uint8)
    }
    labels = {
        'train': np.load(BASE_DIR +'classtrainlabels.npy').astype(np.uint8),
        'val': np.load(BASE_DIR +'classvallabels.npy').astype(np.uint8),
        'test': np.load(BASE_DIR +'classtestlabels.npy').astype(np.uint8)
    }
    return images, labels

class NeuroDataset(Dataset):
    def __init__(self, stage='train', transform=None):
        images, labels = get_task_data_in_numpy()
        self.data = images[stage]
        self.labels = labels[stage]
        self.concat = np.concatenate((self.data[:, np.newaxis, :, :], self.labels[:,
np.newaxis, :, :]), axis=1)
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        concat = self.concat[idx].transpose(1,2,0)
        if self.transform:
            concat = self.transform(concat)
        return concat[0, np.newaxis, :, :], (concat[1, :, :] * 255).long()

class NeuroDataModule(pl.LightningDataModule):
    RESIZE_SHAPE = [224, 224]
    RESIZE_CROP_SHAPE = [224, 224]

    def __init__(self, batch_size=32):
        super().__init__()
        self.batch_size = batch_size
        train_transforms = Compose([
            ToTensor(),
            RandomVerticalFlip(p=0.5),
            RandomHorizontalFlip(p=0.5),
            RandomRotation((0,180)),
        ])
        val_transforms = Compose([
            ToTensor(),
        ])
        self.neuro_train = NeuroDataset('train', transform=train_transforms)
        self.neuro_val = NeuroDataset('val', transform=val_transforms)
        self.neuro_test = NeuroDataset('test', transform=val_transforms)

    def setup(self, stage=None):
        pass

    def train_dataloader(self):
        return DataLoader(self.neuro_train, batch_size=self.batch_size, num_workers=6,
shuffle=False)

    def val_dataloader(self):
        return DataLoader(self.neuro_val, batch_size=self.batch_size)

    def test_dataloader(self):
        return DataLoader(self.neuro_test, batch_size=self.batch_size)


if __name__ == "__main__":
    images, labels = get_task_data_in_numpy()
    loader = NeuroDataModule(500).train_dataloader()
    for i, (img, mask) in enumerate(loader):
        if i == 0:
            img0 = img[0, 0].numpy()
            img1 = img[400, 0].numpy()
            mask = mask[0].numpy()
            plt.figure(figsize=(15, 5))
            plt.subplot(1,4,1)
            plt.imshow(images['train'][0], cmap='gray')
            plt.title("Original Image")
            plt.axis('off')
            plt.subplot(1,4,2)
            plt.imshow(img0, cmap='gray')
            plt.title("After Augmentation")
            plt.axis('off')

            plt.subplot(1,4,3)
            plt.imshow(images['train'][400], cmap='gray')
            plt.title("Original Image")
            plt.axis('off')
            plt.subplot(1,4,4)
            plt.imshow(img1, cmap='gray')
            plt.title("After Augmentation")
            plt.axis('off')


            plt.savefig('asdasd.png')
            break
    # for key in images.keys():
        # print(key, images[key].shape[0], np.unique(labels[key], return_counts=True))
        # print(key, np.std(images[key]), np.mean(images[key]))
        # plt.hist(images[key][0].reshape(256, -1), 256, [0, 256])
        # plt.savefig(f'{key}_hist.png')
    # plt.figure(figsize=(25, 8))
    # for i, index in enumerate([0, 200, 800, 1000]):
    #     img = images['train'][index]
    #     label = labels['train'][index]
    #     img, anno, anno_pred = draw_mask_comparsion(img, label, label)
    #     # breakpoint()
    #     plt.subplot(2, 6, i + 1)
    #     plt.imshow(img)
    #     plt.axis('off')
    #     plt.title(f'Original Image {index}')
    #     plt.subplot(2, 6, i + 1 + 6)
    #     plt.imshow(anno_pred)
    #     plt.axis('off')
    #     plt.title(f'Ground Truth {index}')
    # plt.savefig('test.png')
```

```python
# utils.py
import matplotlib.pyplot as plt
import numpy as np


def draw_mask(img, mask):
    """Draws mask on an image.
    """
    mask = mask.astype(np.uint8)
    img = np.copy(img)
    h, w = img.shape[:2]
    div_factor = 255 / np.max(img)
    for i in range(h):
        for j in range(w):
            if mask[i, j] == 1:
                img[i, j] = np.array([247, 86, 124], dtype=np.float32) / div_factor
            elif mask[i, j] == 2:
                img[i, j] = np.array([153, 225, 217], dtype=np.float32) / div_factor
            elif mask[i, j] == 3:
                img[i, j] = np.array([238, 227, 171], dtype=np.float32) / div_factor
    return img

def draw_mask_comparsion(img, mask, mask_pred):
    """Draws mask on an image with ground truth and prediction.
    """
    h, w= img.shape[:2]
    img = np.repeat(img.reshape(h, w, 1), 3, axis=2)
    annotated = draw_mask(img, mask)
    annotated_pred = draw_mask(img, mask_pred)
    return (img, annotated, annotated_pred)

def save_result(img, annotated, annotated_pred, filename):
    plt.figure(figsize=(15, 5))
    plt.subplot(1, 3, 1)
    plt.imshow(img)
    plt.title('Original')
    plt.subplot(1, 3, 2)
    plt.imshow(annotated)
    plt.title('Ground Truth')
    plt.subplot(1, 3, 3)
    plt.imshow(annotated_pred)
    plt.title('Prediction')
    plt.savefig(filename)

if __name__ == '__main__':
    images = np.load('downloads/task2_2D_4classtestimages.npy')
    masks = np.load('downloads/task2_2D_4classtestlabels.npy')
    img = images[0]
    mask = masks[0]
    mask_pred = masks[0]
    h, w= img.shape
    img = np.repeat(img.reshape(h, w, 1), 3, axis=2)
    annotated = draw_mask(img, mask)
    annotated_pred = draw_mask(img, mask_pred)
    plt.figure(figsize=(15, 5))
    plt.subplot(1, 3, 1)
    plt.imshow(img)
    plt.title('Original')
    plt.subplot(1, 3, 2)
    plt.imshow(annotated)
    plt.title('Ground Truth')
    plt.subplot(1, 3, 3)
    plt.imshow(annotated_pred)
    plt.title('Prediction')
    plt.savefig('annotated.png')
```

```python
# model.py
import torch
import torch.optim as optim
import pytorch_lightning as pl
import segmentation_models_pytorch as smp
from config import *
from utils import *
from sklearn.metrics import classification_report

class SegmentationModule(pl.LightningModule):

    def __init__(self, model, loss_type, encoder, encoder_weights, step_lr):
        super().__init__()
        self.save_hyperparameters()
        if loss_type == 'focal':
            self.loss = smp.losses.FocalLoss(LOSS_MODE)
        elif loss_type == 'dice':
            self.loss = smp.losses.DiceLoss(LOSS_MODE)
        elif loss_type == 'ce':
            self.loss = torch.nn.CrossEntropyLoss(weight=torch.tensor([0.1, 0.3, 0.3, 0.2]))
        else:
            raise NotImplementedError
        if model == 'unet':
            self.model = smp.Unet(
                            encoder_name=encoder,
                            encoder_weights=encoder_weights,
                            in_channels=1,
                            classes=4,
                        )
        elif model == 'fpn':
            self.model = smp.FPN(
                            encoder_name=encoder,
                            encoder_weights=encoder_weights,
                            in_channels=1,
                            classes=4,
                        )
        elif model == 'linknet':
            self.model = smp.Linknet(
                            encoder_name=encoder,
                            encoder_weights=encoder_weights,
                            in_channels=1,
                            classes=4,
                        )

    def predict(self, x):
        with torch.no_grad():
            pred = self.forward(x)
        return pred.argmax(dim=1)

    def forward(self, x):
        x = self.model(x)
        return torch.softmax(x, dim=1)

    def training_step(self, batch, batch_idx):
        x, y = batch # N * C * H * W, N * H * W
        y_hat = self.forward(x)
        loss = self.loss(y_hat, y)
        return {'test_loss': loss, 'y_hat': y_hat.argmax(dim=1), 'y': y.clone().detach()}

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.forward(x)
        loss = self.loss(y_hat, y)
        return {'val_loss': loss, 'y_hat': y_hat.argmax(dim=1), 'y': y.clone().detach()}

    def test_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.forward(x)
        loss = self.loss(y_hat, y)
        return loss

    def validation_epoch_end(self, val_batch_outputs):
        loss = torch.stack([x['val_loss'] for x in val_batch_outputs]).mean()
        y_hat = torch.cat([x['y_hat'] for x in val_batch_outputs], dim=0).flatten()
        y = torch.cat([x['y'] for x in val_batch_outputs], dim=0).flatten()
        print(classification_report(y.cpu().numpy(), y_hat.cpu().numpy()))


    def test_epoch_end(self, test_batch_outputs):
        loss = torch.stack([x['test_loss'] for x in test_batch_outputs]).mean()
        y_hat = torch.cat([x['y_hat'] for x in test_batch_outputs], dim=0).flatten()
        y = torch.cat([x['y'] for x in test_batch_outputs], dim=0).flatten()
        print(classification_report(y.cpu().numpy(), y_hat.cpu().numpy()))

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=3e-2)
        return optimizer
```

```python
import pytorch_lightning as pl
from pytorch_lightning.strategies import DDPStrategy
from model import SegmentationModule
from data import NeuroDataModule
from utils import draw_mask_comparsion, save_result
from config import *
import numpy as np
import matplotlib.pyplot as plt
import torch

torch.manual_seed(42)
np.random.seed(42)

def train(model, loss_type, encoder, encoder_weights, step_lr, gpu_id):
    xd = NeuroDataModule(64)
    model = SegmentationModule(model, loss_type, encoder, encoder_weights, step_lr)
    ddp = DDPStrategy(process_group_backend="nccl", find_unused_parameters=False)
    EPOCH = 100
    if DEBUG:
        train_trainer = pl.Trainer(accelerator='gpu',
                                   devices=[gpu_id],
                                   auto_select_gpus=True,
                                   max_epochs=EPOCH,
                                   check_val_every_n_epoch=5,
                                   log_every_n_steps=10,
                                   )
    else:
        train_trainer = pl.Trainer(accelerator='gpu',
                                   devices=6,
                                   max_epochs=EPOCH,
                                   strategy=ddp,
                                   check_val_every_n_epoch=5,
                                   log_every_n_steps=10,
                                   )
    train_trainer.fit(model=model, datamodule=xd)

def train_one(checkpoint_path):
    xd = NeuroDataModule(32)
    model = SegmentationModule.load_from_checkpoint(checkpoint_path)
    test_trainer = pl.Trainer(accelerator='gpu',
                                   devices=1,
                                   max_epochs=1,
                                   reload_dataloaders_every_n_epochs=2,
                                   log_every_n_steps=10,
                                   )
    test_trainer.fit(model=model, datamodule=xd)

def test(checkpoint_path):
    xd = NeuroDataModule(32)
    model = SegmentationModule.load_from_checkpoint(checkpoint_path)
    test_trainer = pl.Trainer(accelerator='gpu',
                                   devices=1,
                                   max_epochs=1,
                                   check_val_every_n_epoch=2,
                                   reload_dataloaders_every_n_epochs=2,
                                   log_every_n_steps=10,
                                   )
    test_trainer.test(model=model, datamodule=xd)

def validate(checkpoint_path):
    xd = NeuroDataModule(32)
    model = SegmentationModule.load_from_checkpoint(checkpoint_path)
    test_trainer = pl.Trainer(accelerator='gpu',
                                   devices=1,
                                   max_epochs=1,
                                   check_val_every_n_epoch=2,
                                   reload_dataloaders_every_n_epochs=2,
                                    log_every_n_steps=10,
                                    )
    test_trainer.validate(model=model, datamodule=xd)


def visualizeResult(checkpoint_path, save_path):
    sample_indexs = [0, 50, 100, 150, 199]
    xd = NeuroDataModule(32)
    model = SegmentationModule.load_from_checkpoint(checkpoint_path)
    model.eval()
    dataloader = xd.neuro_test
    annotation_matrix = []
    for idx, batch in enumerate(dataloader):
        if idx not in sample_indexs:
            continue
        x, y = batch
        x = x.reshape(1, 1, 256, 256)
        y_hat = model.predict(x)
        x = x.reshape(256, 256).numpy()
        y = y.reshape(256, 256).numpy()
        y_hat = y_hat.reshape(256, 256).numpy()
        img, anno, anno_pred = draw_mask_comparsion(x, y, y_hat)
        pad_img = np.pad(img, [[1,1], [1,1], [0,0]], 'constant', constant_values=1.0)
        pad_anno = np.pad(anno, [[1,1], [1,1], [0,0]], 'constant', constant_values=1.0)
        pad_anno_pred = np.pad(anno_pred, [[1,1], [1,1], [0,0]], 'constant',
constant_values=1.0)
        annotation_matrix.append(np.concatenate([pad_img, pad_anno, pad_anno_pred], axis=1))
    annotation_matrix = np.concatenate(annotation_matrix, axis=0)
    plt.figure(figsize=(len(sample_indexs) * 15, 3 * 15))
    plt.imshow(annotation_matrix)
    plt.axis('off')
    plt.savefig(f'combined_version{save_path}.png')
if __name__ == '__main__':
    # training
    model = 'unet'
    encoder = 'resnet18'
    encoder_weights = 'imagenet'
    loss_type = 'ce'
    step_lr = None
    gpu_id = 1
    train(model, loss_type, encoder, encoder_weights, step_lr, gpu_id)
    # validation, testing and visualization
    for i in [3]:
        print('=' * 20, f'version {i}', '=' * 20)
        checkpoint_path = f'lightning_logs/version_{i}/checkpoints/epoch=99-step=1700.ckpt'
        validate(checkpoint_path)
        test(checkpoint_path)
        visualizeResult(checkpoint_path, i)
```